RAPPORT DU PROJET PET-RESCUE

Jules Cherion et Yoan Rougeolle

Introduction

Ce projet a été très intéressant à faire, puisqu'il nous a permis d'utiliser toutes nos connaissances de java, pour bâtir un petit jeu. Cela n'a pas été du tout facile à faire. Nous allons détailler dans les parties qui suivent les différentes étapes clés de ce projet ainsi que nos difficultés.

La Modélisation du Modèle

Nous avons tout d'abord commencé à réfléchir à la modélisation du modèle qui n'a pas été si simple. Le début d'un projet est toujours difficile puisqu'on ne sait pas par où commencer. Mais notre objectif a été de faire un début de modèle avec un affichage dans le terminal pour vérifier que ça marche.

Pour faire la modélisation nous avons dû analyser le jeu de base (que peut faire le joueur, quelles sont les différentes phases de jeu, quelles classes faire, quelles relations entre les classes). Très vite on a décidé de faire une classe **Plateau** contenant un tableau de 2 dimensions de cases (les cases sont représentées par la classe **Cell**).

On avait d'abord prévu que les cases contiendraient les blocs, animaux, murs, mais on a décidé que ce serait plus simple d'utiliser l'héritage. Un objet de classe **Cell** représente une case vide, mais si cette classe est héritée (par les classes **Bloc**, **Mur**, **Pet**) alors cela représente le **Bloc/Mur/Pet**.

Après cette première modélisation, qui est restée jusqu'à la fin, il fallait commencer à l'implémenter

Le Début de l'Implémentation

Pour l'implémentation, on a commencé avec les classes **Cell** et **Bloc**. Dans **Cell**, on a mis des attribut *i* et *j* qui représentent respectivement la ligne et la colonne de la **Cell** dans un plateau. Ces attributs étaient prévus pour l'interface graphique.

En effet, au début les fonctions d'affichage (pour le terminal et pour l'interface graphique) étaient prévus dans Cell et Plateau ce qui a été le cas pendant longtemps. On avait prévu de donner un objet **Graphics** à la méthode d'affichage, afin d'avoir juste à utiliser la méthode *fillRect()* avec les attributs *i* et *j* pour dessiner un rectangle à l'endroit voulu.

La Classe **Bloc** contient un attribut pour la couleur, et un attribut pour savoir s'il est vide, c'est-à-dire s'il a été détruit.

Ensuite on a commencé à faire la classe **Plateau**. Elle contient un tableau de 2 dimensions de **Cell**. Pour faciliter la création de niveau, on a tout de suite créer le constructeur qui à partir d'un fichier *.txt*, qui construit le tableau de **Cell** avec les bons blocs.

Après nous avons ajouté la fonction qui détruit les ensembles de blocs de même couleur en indiquant la case. On utilise de la récursion pour cela, et on a dû ajouter une fonction *explose()* dans **Bloc**. Mais pour faciliter l'écriture de la fonction *explose()* dans **Plateau**, on a également ajouté cette fonction qui ne fait rien dans **Cell**, afin que *explose()* de **Bloc** soit une redéfinition. Au départ la fonction *explose()* ne renvoyait rien, mais on a décidé qu'il fallait qu'elle compte également le nombre de blocs détruits pour qu'il soit utilisé plus tard pour calculer le score.

Enfin on a ajouté toutes les fonctions qui vont permettre de faire chuter les blocs et on a rajouté les **Mur**, puisqu'ils ont une importance dans la chute.

La Première Difficulté

Après avoir fait la fonction qui décale les blocs vers le bas, il fallait faire la fonction qui décale les blocs vers la gauche. Ceci a été assez compliqué. En effet, on ne comprenait pas du tout ce comportement dans le vrai jeu, et on voyait qu'il y avait beaucoup de cas particuliers. Puis après une semaine de réflexion, on a réussi à comprendre comment cela marchait. Le fonctionnement de la méthode est décrit en commentaire dans le code, mais on comprenait intuitivement pourquoi à tel moment les blocs se décalaient vers la gauche, mais en déduire l'algorithme derrière a été vraiment compliqué.

Fin de l'Implémentation du Modèle

Étant bloqué sur la fonction qui décale les blocs vers la gauche, on a décidé d'ajouter les animaux (la classe **Pet**). Ceci n'a pas été très compliqué. On a du ajouter un fonction pour sauver les animaux une fois qu'ils étaient sur la dernière ligne puis on a modifié la constructeur de **Plateau** pour qu'il détecte les animaux, on a ajouté un attribut dans **Plateau** pour en savoir le nombre, puis on a pu faire une fonction pour savoir si la partie était fini (s'il n'y a plus d'animaux ou si le joueur ne peut plus jouer). Une fois fait, la partie de l'implémentation du modèle était terminée et n'a presque pas été changée depuis.

La Version Textuelle du Jeu

Une fois le modèle fonctionnel, il fallait intégrer le code permettant le déroulement d'une partie, l'interaction avec un joueur, l'affichage. A ce moment-là, lorsqu'on implémentait le version textuelle du jeu, il fallait également prévoir la future interface graphique.

Pour modéliser cela il fallait une classe pour représenter un joueur, une classe pour représenter l'affichage, une classe pour représenter l'interaction et une classe permettant la communication entre tous ces éléments.

- Pour les joueurs on a créé la classe **Joueur** qui ne contenait que les attributs *nom* et *score*.
- Pour l'affichage, on s'est dit qu'il fallait faire une interface **Afficheur** pour prévoir l'interface graphique et on a créé la classe **TerminalAfficheur** responsable de l'affichage dans le terminal, qui implémente **Afficheur**.
- Pour l'interaction avec le joueur on a fait à peu près la même chose (une interface **Interacteur** et un classe **TerminalInteracteur** qui implémente **Interacteur**).

Après on a créé la classe **Jeu** qui contient un **Plateau**, un Joueur, un **Afficheur** et un Interacteur. Dans **TerminalAfficheur** on a ajouté la fonction pour afficher le plateau (qui appelait la méthode dans **Plateau** pour afficher le plateau, ceci a été changé beaucoup plus tard), des fonctions pour afficher le score et la fin de partie. Dans **TerminalInteracteur** on a ajouté toutes fonctions qui permettent de récupérer les inputs du Joueur à l'aide de la classe **Scanner**.

Enfin dans **Jeu** on a ajouté une fonction qui demande quelle interface le joueur veut utiliser. On a ajouté une grosse fonction qui s'occupait du déroulement du jeu (tant que le jeu n'était pas terminé, on demandait dans quelle case le joueur voulait jouer grâce à l'interacteur, on mettait à jour le score, puis on déplaçait les blocs, puis on sauvait les animaux, puis on déplaçait les blocs, etc...).

Le Robot

Un fois que cette version textuelle marchait bien, on a décidé d'ajouter le robot. Le problème c'est que le robot ne peut pas utiliser l'interacteur pour jouer
donc on a dû revoir la classe **Joueur** et l'emplacement de l'interacteur. Pour cela on a différencié deux types de joueur: les humains et les robots.

On a donc décidé que la classe **Joueur** serait abstraite (puisque ce n'aurait pas de sens d'instancier un joueur si il n'est ni humain ni robot), de créer deux classes **Humain** et **Robot** qui héritent de **Joueur** et que l'interacteur ne serait plus dans **Jeu** mais dans la classe **Humain**. Grâce à cette implémentation, dans **Jeu**, on peut demander directement au joueur pour savoir quelle case il veut jouer. Dans **Humain** on demande à l'interacteur et dans **Robot** il analyse le plateau et envoie les coordonnées de la case qu'il "veut" jouer. On avait prévu de faire un robot utilisant un algorithme de résolution, cependant nous n'y sommes pas arrivés. Le robot se contente juste de jouer de façon aléatoire.

Ajout de l'Interface Graphique: Le Début

Ensuite il fallait ajouter l'interface graphique. Pour ce faire, on a créé une classe **Visuelle** qui la représente. Cette classe implémente **Afficheur** et **Interacteur**. On a décidé d'implémenter les deux pour simplifier l'accès aux attributs de chacun. **Visuelle** étend la classe **JFrame** et contient des attributs qui représentent chacun des écrans.

En effet, on a décidé de faire une classe pour chacun des écrans et chacune de ces classes étendent **JPanel** (classe **MenuXXX**).

Et là arrive là deuxième grosse difficulté : une interface graphique ne marche pas comme un terminal. En effet dans la classe **Jeu** (celle qui permet le déroulement d'une partie), qui demandait au joueur son nom, quelle case il voulait jouer, était fait dans le terminal avec un **Scanner**, ce qui met en "pause" le programme. C'était **Jeu** qui demandait à l'interacteur l'information. Mais dans une interface graphique c'est tout à fait l'inverse, c'est l'interacteur qui va faire avancer le jeu et pas l'inverse. Alors au tout début on avait fait des boucles *tant que* le joueur n'a pas répondu, alors le programme attends (grâce à des **Thread**.*sleep()*). Mais trouvant cette solution très mauvaise, on a dû changer la fonction qui s'occupait du déroulement du jeu, et on l'a "découpée" en plus petites fonctions que l'interacteur pourrait appeler par la suite.

Par contre un autre problème apparaît, l'interface graphique fonctionne à peu près bien, mais la version textuelle ne marche plus puisque les **Scanner** ne s'ouvrent plus, alors

on a ajouté une fonction *prochainCoup()* dans l'interface **Interacteur** qui permet d'ouvrir un **Scanner** dans la version textuelle, mais dans la version interface graphique, cette fonction ne fait rien.

La Classe VisuPlateau

Après avoir résolu cette difficulté, on a ajouté les différents écrans. Pas grand choses à dire sur les classes **MenuXXX**, à part pour la classe **VisuPlateau**. Cette classe permet l'affichage du plateau dans l'interface graphique. Pour faire cela, on redéfinit la fonction *paintComponent()* de la classe **JPanel** et dans le classe **Plateau**, on avait ajouté une fonction d'affichage qui le prenait le **Graphics** en argument, et c'était les classes **Cell/Bloc/Mur/Pet** qui avait des fonctions d'affichage utilisant le **Graphics** pour dessiner.

Cependant, on a remarqué (beaucoup trop tard) que ces fonctions d'affichage n'avaient rien à faire dans ces classes. Donc on a dû créer une fonction qui permet de cloner un **Plateau**, puis on a déplacé ces fonctions d'affichage dans les classes **TerminalAfficheur** et **VisuPlateau**. On n'aurait jamais dû mettre ces fonctions dans les classes du modèle.

L'Animation

Après avoir fini les différents écrans de l'interface graphique, et régler les soucis avec les fonctions d'affichage du **Plateau**, on s'est dit qu'il faudrait qu'on puisse voir les blocs bouger lorsqu'on joue, qu'il ne se déplace pas instantanément. Assez rapidement, on s'est dit qu'il faudrait calculer de combien de cases, les blocs se déplacent entre chaque mouvement. Ainsi on a ajouté des attributs *xOff* et *yOff* calculant donc de combien de cases s'est déplacé un **Bloc** ou un **Pet**. Après dans **VisuPlateau**, on a ajouté deux tableaux d'entiers représentant le décalage entre la position du bloc entre le début et la fin de l'animation en pixels.

Et pour exécuter l'animation, lorsqu'on demande d'afficher le plateau, cela appelle une fonction dans **Visuelle** qui crée un **Thread** qui appelle toutes les soixantièmes de seconde la fonction *paintComponent()* de **VisuPlateau**. A chaque appelle de la fonction *paintComponent()*, les décalages des blocs se réduisent créant ainsi une animation. L'animation s'arrête lorsqu'il n'y a plus de **Bloc/Pet** qui "bougent". On a décidé d'interdire le joueur de pouvoir jouer durant l'animation.

Mais il y a eu deux petits problèmes. Le premier était qu'on calculait tous les déplacements, puis on déplaçait les **Bloc/Pet**, du coup tout les **Pet** qui étaient censés être sauvés disparaissaient tout de suite, en fait il fallait qu'on décompose plus les mouvements (d'abord bouger les **Bloc/Pet**, puis sauver UN **Pet**, bouger les blocs à nouveau, sauver un **Pet**, ...).

Le deuxième problème était la fluidité de l'animation. Au début nous avons cru que cela venait du fait que l'on calculait trop de choses entre deux affichages du plateau, mais après quelques recherches internet il semble que cela vienne de **JFrame**, et donc en ajoutant une ligne de code avant l'appelle de paintComponent le problème a été réglé.

La Sauvegarde des données du Joueur

Après relecture du sujet, pour vérifier que l'on avait rien oublié, on avait totalement oublié le système de progression d'un joueur. Avant le joueur pouvait faire le niveau 6 avant le niveau 1. Du coup on a dû ajouter cela, ainsi que l'ajout de la sauvegarde des scores. On a donc ajouté un attribut *levelMax* qui représente jusqu'à quel niveau il peut jouer, et un attribut *bestScores* qui est un tableau d'entiers stockant les meilleurs scores du joueur à chaque niveaux.

Enfin il fallait ajouter la sauvegarde/chargement des données du joueur dans un fichier. On a décidé de seulement sauvegarder les données des joueurs humains. Il y a un fichier de sauvegarde par joueur sous la forme *nom.txt* stocké dans le dossier Sauvegarde

Ajout de la fusée

La dernière fonctionnalité ajoutée est la fusée. On a dû ajouter dans **Plateau** une fonction pour enlever une colonne. Au départ, on voulait qu'il y ait un nombre déterminé de fusée pour chaque niveau. Finalement on a décidé qu'il serait mieux que ce soit un nombre déterminé de fusée par joueur.

On a également dû changer les conditions de défaite, puisqu'avant, lorsque le joueur n'avait plus de cases pour jouer, il avait perdu. Il faut également vérifier que le joueur n'a plus de fusée. De plus, si le joueur ne peut plus jouer sur un case mais a des fusées et qu'il ne veut pas les utiliser, alors il faut qu'il puisse abandonner. On a dû ajouter un bouton abandonne sur l'interface graphique et une option abandonne pour la version textuelle.

Enfin la dernière chose à faire a été l'affichage de la fusée sur l'interface graphique. Au départ on voulait qu'il y ait un bouton pour activer l'option fusée, puis un fois activée, une image de fusée suive le curseur de la souris, mais après on a décidé de d'afficher seulement en dessous de la colonne visée par la souris, une image de fusée.

Conclusion

Ce projet était sur certains aspects comme la réalisation du fonctionnement du jeu plus facile que ce qu'on imaginait, mise à part la compréhension de la fonction *left()*. Cependant la notion de vue/modèle vu en cours a été bien plus difficile à mettre en place dû à notre mauvais "point de départ" où nous n'avions pas fait de représentation précise de nos classes et de leur fonctionnement.

Cela nous a permis de comprendre à quelle point il était important de définir en amont l'architecture d'un programme ainsi que la clarté du code afin de pouvoir le modifier et de le comprendre aisément.